Functions

Within any computer programming language, large programs are broken down into modules. These modules may be called subroutines (in COBOL or RPG), procedures (in Pascal), methods (in Java) or functions (in C and C++). When you use functions, your programs are easier to follow because they are organized into reasonably-sized units. Each function is reusable, that is, it is easily separated from one program and included with another as needed. Breaking a program's segments into detachable functions allows many programmers to work on a single project; the individual functions can be assembled into a major working product.

In C++, every function consists of two parts:
- a header
- a body

The body is always contained between a pair of curly brackets.

Every C++ function header consists of three parts:
- a return type
- the function name
- an argument list enclosed in parentheses

For example, here is a main() function:

```
void main()
 {
   cout << "Hello" << endl;
 }
```

Its return type is void, its name is main, and its argument list is void.

As another example, here is a function that returns a value:

```
double GetSalary()
  {
    double salary;
    cout << "Enter your salary ";
    cin >> salary;
    return(salary);
  }
```

This function's name is GetSalary. It takes no arguments, but the header indicates it will return a double value. The function itself declares a double variable named salary. It

prompts the user for a value, then reads the value from the keyboard. Finally the value in the salary variable is returned to any program that calls the GetSalary() function.

One last example -- a function that returns a double and takes a character and an integer as arguments:

```
double ComputePay(char empCode, int hours)
  {
    double payAmount;
     if (empCode == 'A')
        payAmount = 5.50 * hours;
      else
         payAmount = 7.65 * hours;
      return(payAmount);
   }
```

This function uses the character code and the integer hours to perform a pay calculation. The double results returns to any program that calls ComputePay.

Every C++ function can, at most, return a single value.

In order to use a function within a C++ program, you must do two things in addition to writing the function:

- prototype the function
- call the function

The prototype of a function is a model that notifies the program what a function will do - what it will return, what its name will be, and what type of data it expects as an argument. Some examples of prototypes:

```
void PrintMyInitial(char)  // will return nothing, needs a character argument
double CalcPay(int, int);  // will return a double, needs two integers as arguments
char GetCode();     // will return a character, needs nothing as an argument
```

Prototypes, like variables, may be declared globally or locally. If you declare a prototype globally, any functions may call the function. If you declare the prototype locally, only the function that declares the prototype may call the function. Global prototypes should be placed where other global declarations are placed; usually at the beginning of the program before the main() function and after the preprocessor directives. Local function prototypes should be placed inside the function that uses them with the variable declarations and prior to any executable statements.

For example, the following code shows that main() will contain an integer variable and a double function. The double function will require two arguments. The integer declaration and the function prototype could just as well have been listed in reverse order. The function prototype has been declared globally.

```
double ComputeSum(double, double);   // global function prototype
void main()
 {
    int x;
    .
    .
    . /* rest of program */
 }
```

In the situation above, main() can call the ComputeSum() function; in addition, any other functions written after main() may call ComputeSum(). The following example shows ComputeSum() declared locally.

```
void main()
 {
    double ComputeSum(double, double);   // local function prototype
    int x;
    .
    .
    . /* rest of program */
 }
```

When you call a function you use its name and pass it any required arguments. If you want to save the value returned by a function, you must assign it to a variable. Thus, if you have prototyped a functions as: **void PrintMyInitial(char)** then you can call it with **PrintMyInitial('K');** or **PrintMyInitial(someInitial);** if someInitial is a character variable you have defined.

For the function whose prototype is **double CalcPay(int, int);**
you probably want your function call to look like one of the following:

```
answer = CalcPay(hrs, rate);
        /* where answer is a double variable and hrs and rate are integers */
amount = CalcPay(40,12);
        /* where amount is a double variable and you want to use constants*/
        /* for arguments */
cout << CalcPay(myHours,15 << endl;
        /* where you don't want to store the double value returned - */
        /* but you do want to print it */
```

For a function that takes no arguments, like **char GetCode();** you use empty parentheses in the function call, as in **employeeCode = GetCode();** where employeeCode has been defined as a character variable.

All variables declared within a function are local to that function. The variables names you use to pass data to a function may have the same name as variables within the function, but they are not required to have the same names. Consider a function named AddNums:

```
int AddNums(int a, int b)
 {
   int z;
   z = a + b;
   return(z);
 }
```

Now, look at this main() program:

```
void main()
 {
    int a = 5, b = 6, c = 7, d = 44, e, f, g, h;
    int AddNums(int, int);
    e = AddNums(a,b);
    f = AddNums(c,d);
    g = AddNums(12, 88);
    h = AddNums(a,16);
 }
```

The value of e is 11 because a and b (5 and 6) are passed to AddNums. The a and b in AddNums take on the values 5 and 6 respectively, and 11 is returned to e.

The value of f is 51 because c and d (7 and 44) are passed to AddNums where they become a and b. The result is returned to f.

Similarly, g's value is 100 and h's value is 21.

Functions provide a high level of security for data. No matter what happens to variables within a function, no variables in any other function are affected unless you expressly want them to be.

For example, consider this function:

```
void SomeFunction(int y)
 {
   y = 55;
   return;
  }
```

If you write a main() program like this:

```
void SomeFunction(int);
void main()
 {
   int y = 2;
   SomeFunction(y);
   cout << y << endl;
  }
```

The output from the final program line is 2, not 55. The y in main() is assigned the value 2. When y is passed to SomeFunction, the y in SomeFunction takes on the value 2. Then the y in SomeFunction is assigned 55. However, nothing is returned from SomeFunction, so the y in main() is still 2. If you want SomeFunction to alter y, it must be written like this:

```
int SomeFunction(int y)
 {
  y = 55;
  return(y);
 }
```

In main(), the function call you want is **y = SomeFunction(y);** which passes the value of y to the function, and assigns the returned value back into main()'s variable y.

You can cause a function to alter a variable's value without returning anything -- if you pass the variable's memory address to the function. In C++, the easiest way to do this is to declare the parameter as a reference variable in the function header. The reference variable receives the address of the variable that was passed and is considered to be an “alias” for the original variable. For example - consider this function:

```
void GiveMeAddress(int &d)
 {
  d = 77;
  return;
 }
```

If you write a main() that passes a variable to a function that accepts a reference variable, then the function "knows where the variable lives" and can alter it directly: In the following program, the last line prints 77 because p has been altered by the GiveMeAddress function in which d had the same memory address as p in main().

```
void GiveMeAddress(int *);
void main()
 {
   int p = 3;
   GiveMeAddress(&p);
   cout << p < endl;
 }
```